

Algorithms and Data Structures

C3. Disjoint-set Data Structure/Union-Find

Gabriele Röger and Patrick Schneider

University of Basel

April 30, 2026

1 / 26

Algorithms and Data Structures

April 30, 2026 — C3. Disjoint-set Data Structure/Union-Find

C3.1 Union-Find

C3.2 Connected Components and Equivalence Classes

C3.3 Summary

2 / 26

C3. Disjoint-set Data Structure/Union-Find

Union-Find

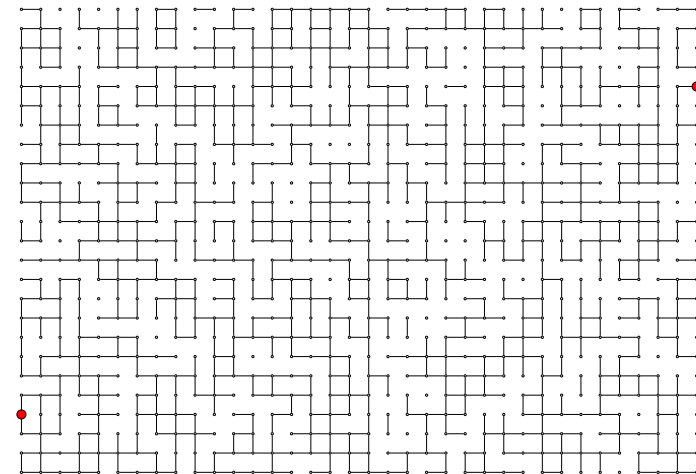
C3.1 Union-Find

3 / 26

C3. Disjoint-set Data Structure/Union-Find

Union-Find

Questions



Are the red vertices connected?
How many connected components does the graph have?

4 / 26

Connected Components as Disjoint Sets

Set of conn. components as collection of disjoint sets of objects.

- ▶ One set for all vertices of one connected component.
- ▶ Operations:
 - ▶ **Union:** Given two objects, merge the sets that contain them into one.
Introduce a new edge between the given vertices, connecting their connected components.
 - ▶ **Find:** Given an object, return a representative of the set that contains it.
Given a vertex, return a representative vertex for its connected component.
 - ▶ Must return the same representative for all objects in the set.
 - ▶ The representative may only change if set gets merged.
 - ▶ Two objects are in the same set (two vertices are connected) if **find** returns the same representative for them.
 - ▶ **Count:** Return the number of sets
Return the number of connected components.

Union-Find Data Type

```

1 class UnionFind:
2     # Initialization for n objects (with names 0, ..., n-1).
3     def __init__(n: int) -> None
4
5     # Merge the sets containing objects v and w.
6     def union(v: int, w: int) -> None
7
8     # Representative for set containing v.
9     # May change if set is merged by call of union,
10    # but not otherwise.
11    def find(v: int) -> int
12
13    # Number of sets.
14    def count() -> int

```

(Somewhat) Naive Algorithm: Quick-Find

- ▶ For n objects: Array **representative** of length n .
- ▶ Entry at position i is representative of the set containing i .
- ▶ Initially, every object is (alone) in its own set, and thus its representative.
- ▶ Update the array in every call of **union**.

Quick-Find Data Structure

```

1 class QuickFind:
2     def __init__(self, no_nodes):
3         self.components = no_nodes
4         self.representative = list(range(no_nodes))
5
6     def count(self):
7         return self.components
8
9     def find(self, v):
10        return self.representative[v]

```

[0, 1, ..., no_nodes-1]

Quick-Find Data Structure (Continued)

```

20 def union(self, v, w):
21     repr_v = self.find(v)
22     repr_w = self.find(w)
23     if repr_v == repr_w: # already in same component
24         return
25     # replace all occurrences of repr_v in
26     # self.representative with repr_w
27     for i in range(len(self.representative)):
28         if self.representative[i] == repr_v:
29             self.representative[i] = repr_w
30     self.components -= 1 # we merged two components

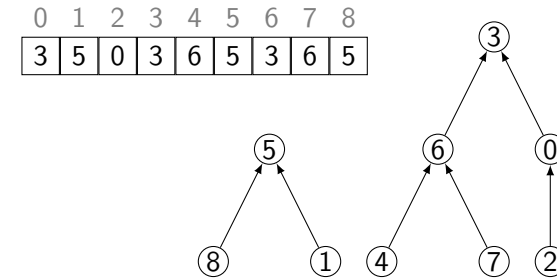
```

Running time?

- ▶ Cost model = number of array accesses
- ▶ one access for every call of find
- ▶ between $n + 3$ and $2n + 1$ accesses for every call of union that merges two components

Better: Quick-Union aka Disjoint-set Forest

- ▶ (implicit) tree for representing each set
- ▶ represented as array with parent nodes as entries (root: reference to itself)



- ▶ Root node serves as representative of the set.

Quick-Union Data Structure

```

1 class QuickUnion:
2     def __init__(self, no_nodes):
3         self.parent = list(range(no_nodes))
4         self.components = no_nodes
5
6     def find(self, v):
7         while self.parent[v] != v:
8             v = self.parent[v]
9         return v
10
11    def union(self, v, w):
12        repr_v = self.find(v)
13        repr_w = self.find(w)
14        if repr_v == repr_w: # already in same component
15            return
16        self.parent[repr_v] = repr_w
17        self.components -= 1
18
19    # count as in QuickFind

```

First Improvement

- ▶ **Problem with Quick-Union:** Trees can degenerate into chains. → find requires linear time in the size of the set.
- ▶ **Idea:** In union the root of the tree with lower height becomes a child of the root of the higher tree.

Ranked Quick-Union Algorithm

```

1 class RankedQuickUnion:
2     def __init__(self, no_nodes):
3         self.parent = list(range(no_nodes))
4         self.components = no_nodes
5         self.rank = [0] * no_nodes # [0, ..., 0]
6
7     def union(self, v, w):
8         repr_v = self.find(v)
9         repr_w = self.find(w)
10        if repr_v == repr_w:
11            return
12        if self.rank[repr_w] < self.rank[repr_v]:
13            self.parent[repr_w] = repr_v
14        else:
15            self.parent[repr_v] = repr_w
16            if self.rank[repr_v] == self.rank[repr_w]:
17                self.rank[repr_w] += 1
18        self.components -= 1
19
20    # connected, count and find as in QuickUnion

```

Second Improvement

Path Compression

- ▶ **Idea:** During `find`, reconnect all traversed nodes to the root.
- ▶ We do not update the height of the tree during path compression.
 - ▶ Value of `rank` can deviate from the actual height.
 - ▶ That's why it is called **rank** and not height.

Ranked Quick-Union Algorithm with Path Compression

```

1 class RankedQuickUnionWithPathCompression:
2     def __init__(self, no_nodes):
3         self.parent = list(range(no_nodes))
4         self.components = no_nodes
5         self.rank = [0] * no_nodes # [0, ..., 0]
6
7     def find(self, v):
8         if self.parent[v] == v:
9             return v
10        root = self.find(self.parent[v])
11        self.parent[v] = root
12        return root
13
14    # connected, count and union as in RankedQuickUnion

```

Discussion

- ▶ With all improvements, we achieve **almost constant amortized cost** for all operations.
- ▶ **More precisely:** [Tarjan 1975]
 - ▶ m calls of `find` for n objects (and at most $n - 1$ calls of `union`, merging two components)
 - ▶ $O(m\alpha(m, n))$ array accesses
 - ▶ α is inverse of a variant of the **Ackermann function**
 - ▶ In practice $\alpha(m, n) \leq 3$.
- ▶ **Nevertheless:** there cannot be a union-find structure that guarantees linear running time. (under cell-probe model, only accounting for memory access)

Comparison to Exploration-based Approach

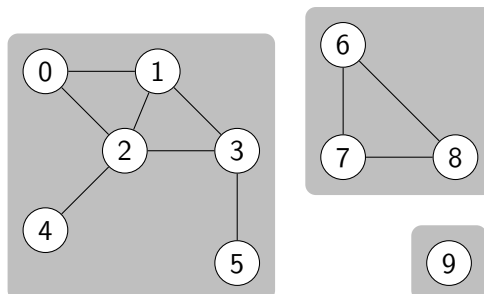
- ▶ Chapter C2: Algorithm `ConnectedComponents`, based on **graph exploration**.
- ▶ After the precomputation, queries only require constant time.
- ▶ In practise, disjoint-set forests are often faster, because for many applications, we do not have to build up the full graph.
- ▶ If the graph has already been built up, graph exploration can be better.
- ▶ Another advantage of union find:
 - ▶ **Dynamic** approach
 - ▶ We can easily introduce further edges.

C3.2 Connected Components and Equivalence Classes

Reminder: Connected Components

Undirected graph

- ▶ Two vertices u and v are in the same **connected component** if there is a path between u and v (= vertices u and v are **connected**).



Connected Components: Properties

- ▶ The connected components define a **partition** of the vertices:
 - ▶ Every vertex is in a connected component.
 - ▶ No vertex is in more than one connected component.
- ▶ “is connected with” is an **equivalence relation**.
 - ▶ **reflexive**: Every vertex is connected with itself.
 - ▶ **symmetric**: If u is connected with v , then v is connected with u .
 - ▶ **transitive**: If u is connected with v , and v with w , then u is connected with w .

Partition in General

Definition (Partition)

A **partition** of a finite set M is a set P of non-empty subsets of M , such that

- ▶ every element of M is in some set in P :
 $\bigcup_{S \in P} S = M$, and
- ▶ that sets in P are pairwise disjoint:
 $S \cap S' = \emptyset$ for $S, S' \in P$ with $S \neq S'$.

The sets in P are called **blocks**.

$$M = \{e_1, \dots, e_5\}$$

- ▶ $P_1 = \{\{e_1, e_4\}, \{e_3\}, \{e_2, e_5\}\}$ is a partition of M .
- ▶ $P_2 = \{\{e_1, e_4, e_5\}, \{e_3\}\}$ is not a partition of M .
- ▶ $P_3 = \{\{e_1, e_4, e_5\}, \{e_3\}, \{e_2, e_5\}\}$ is not a partition of M .
- ▶ $P_4 = \{\{e_1\}, \{e_2\}, \{e_3\}, \{e_4\}, \{e_5\}\}$ is a partition of M .

Equivalence Relations in General

Definition (Equivalence Relation)

An **equivalence relation** over set M is a **symmetric, transitive and reflexive** relation $R \subseteq M \times M$.

We write $a \sim b$ for $(a, b) \in R$ and say that **a is equivalent to b** .

- ▶ **symmetric**: $a \sim b$ implies $b \sim a$
- ▶ **transitive**: $a \sim b$ and $b \sim c$ implies $a \sim c$
- ▶ **reflexive**: for all $e \in M$: $e \sim e$

Equivalence Classes

Definition (Equivalence Classes)

Let R be an equivalence relation over M .

The **equivalence class** of $a \in M$ is the set

$$[a] = \{b \in M \mid a \sim b\}.$$

- ▶ The set of all equivalence classes is a partition of M .
- ▶ **Vice versa**:
 For partition P define $R = \{(x, y) \mid \exists B \in P : x, y \in B\}$
 (i.e. $x \sim y$ if and only if x and y are in the same block).
 Then R is an equivalence relation.
- ▶ We can consider blocks in partitions as equivalence classes and vice versa.

Union-Find and Equivalences

- ▶ **Given**: finite set M ,
 sequence s of equivalences $a \sim b$ over M
- ▶ Consider equivalences as edges in a graph with set M of vertices.
- ▶ The connected components correspond to the equivalence classes of the **finest equivalence relation** that considers all equivalences from s .
 - ▶ no “unnecessary” equivalences.

Can use **union-find data structures** to **determine equivalence classes**.

C3.3 Summary

Summary

- ▶ A **union-find** data structure maintains a collection of **disjoint sets**.
 - ▶ **union**: merge two sets.
 - ▶ **find**: identify the set containing an object and return its representative.
- ▶ Good implementation: **Disjoint-set forest** with improvements to keep the height of the trees low:
 - ▶ Union adjoins the shorter tree to the taller tree.
 - ▶ Find reconnects traversed nodes to the root (**path compression**).
- ▶ Applications:
 - ▶ Connected components
 - ▶ Finest equivalence relation